

Introduction

Welcome to **CS188 - Data Science Fundamentals!** We plan on having you go through some grueling training so you can start crunching data out there... in today's day and age "data is the new oil" or perhaps "snake oil" nonetheless, there's a lot of it, each with different purity (so pure that perhaps you could feed off it for a life time) or dirty which then at that point you can either decide to dump it or try to weed out something useful (that's where they need you...)

In this project you will work through an example project end to end.

Here are the main steps:

1. Get the data
2. Visualize the data for insights
3. Preprocess the data for your machine learning algorithm
4. Select a model and train
5. Does it meet the requirements? Fine tune the model



Working with Real Data

It is best to experiment with real-data as opposed to aritifical datasets.

There are many different open datasets depending on the type of problems you might be interested in!

Here are a few data repositories you could check out:

- [UCI Datasets \(<http://archive.ics.uci.edu/ml/>\)](http://archive.ics.uci.edu/ml/)
- [Kaggle Datasets \(\[kaggle.com\]\(https://www.kaggle.com\)\)](https://www.kaggle.com)
- [AWS Datasets \(<https://registry.opendata.aws>\)](https://registry.opendata.aws)

Below we will run through an California Housing example collected from the 1990's.

Setup

In [1]:

```
import sys
assert sys.version_info >= (3, 5) # python>=3.5
import sklearn
assert sklearn.__version__ >= "0.20" # sklearn >= 0.20

import numpy as np #numerical package in python
import os
%matplotlib inline
import matplotlib.pyplot as plt #plotting package

# to make this notebook's output identical at every run
np.random.seed(42)

#matplotlib magic for inline figures
%matplotlib inline
import matplotlib # plotting library
import matplotlib.pyplot as plt

# Where to save the figures
ROOT_DIR = "."
IMAGES_PATH = os.path.join(ROOT_DIR, "images")
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_name, tight_layout=True, fig_extension="png", resolution=300):
    """
    plt.savefig wrapper. refer to
    https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.savefig.html
    """
    path = os.path.join(IMAGES_PATH, fig_name + "." + fig_extension)
    print("Saving figure", fig_name)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

In [2]:

```
import os
import tarfile
import urllib
DATASET_PATH = os.path.join("datasets", "housing")
```

Intro to Data Exploration Using Pandas

In this section we will load the dataset, and visualize different features using different types of plots.

Packages we will use:

- [Pandas \(<https://pandas.pydata.org>\)](https://pandas.pydata.org): is a fast, flexible and expressive data structure widely used for tabular and multidimensional datasets.
- [Matplotlib \(<https://matplotlib.org>\)](https://matplotlib.org): is a 2d python plotting library which you can use to create quality figures (you can plot almost anything if you're willing to code it out!)
 - other plotting libraries:[seaborn \(<https://seaborn.pydata.org>\)](https://seaborn.pydata.org), [ggplot2 \(<https://ggplot2.tidyverse.org>\)](https://ggplot2.tidyverse.org)

In [3]:

```
import pandas as pd

def load_housing_data(housing_path):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

In [4]:

```
housing = load_housing_data(DATASET_PATH) # we load the pandas dataframe
housing.head() # show the first few elements of the dataframe
# typically this is the first thing you do
# to see how the dataframe looks like
```

Out[4]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	med
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	

A dataset may have different types of features

- real valued
- Discrete (integers)
- categorical (strings)

The two categorical features are essentially the same as you can always map a categorical string/character to an integer.

In the dataset example, all our features are real valued floats, except ocean proximity which is categorical.

In [5]:

```
# to see a concise summary of data types, null values, and counts
# use the info() method on the dataframe
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude           20640 non-null float64
latitude            20640 non-null float64
housing_median_age  20640 non-null float64
total_rooms          20640 non-null float64
total_bedrooms       20433 non-null float64
population          20640 non-null float64
households           20640 non-null float64
median_income        20640 non-null float64
median_house_value   20640 non-null float64
ocean_proximity      20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

In [6]:

```
# you can access individual columns similarly
# to accessing elements in a python dict
housing["ocean_proximity"].head() # added head() to avoid printing many columns..
```

Out[6]:

```
0    NEAR BAY
1    NEAR BAY
2    NEAR BAY
3    NEAR BAY
4    NEAR BAY
Name: ocean_proximity, dtype: object
```

In [7]:

```
# to access a particular row we can use iloc
housing.iloc[1]
```

Out[7]:

```
longitude           -122.22
latitude            37.86
housing_median_age  21
total_rooms          7099
total_bedrooms       1106
population          2401
households           1138
median_income        8.3014
median_house_value   358500
ocean_proximity      NEAR BAY
Name: 1, dtype: object
```

In [8]:

```
# one other function that might be useful is
# value_counts(), which counts the number of occurrences
# for categorical features
housing["ocean_proximity"].value_counts()
```

Out[8]:

```
<1H OCEAN      9136
INLAND        6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND          5
Name: ocean_proximity, dtype: int64
```

In [9]:

```
# The describe function compiles your typical statistics for each
# column
housing.describe()
```

Out[9]:

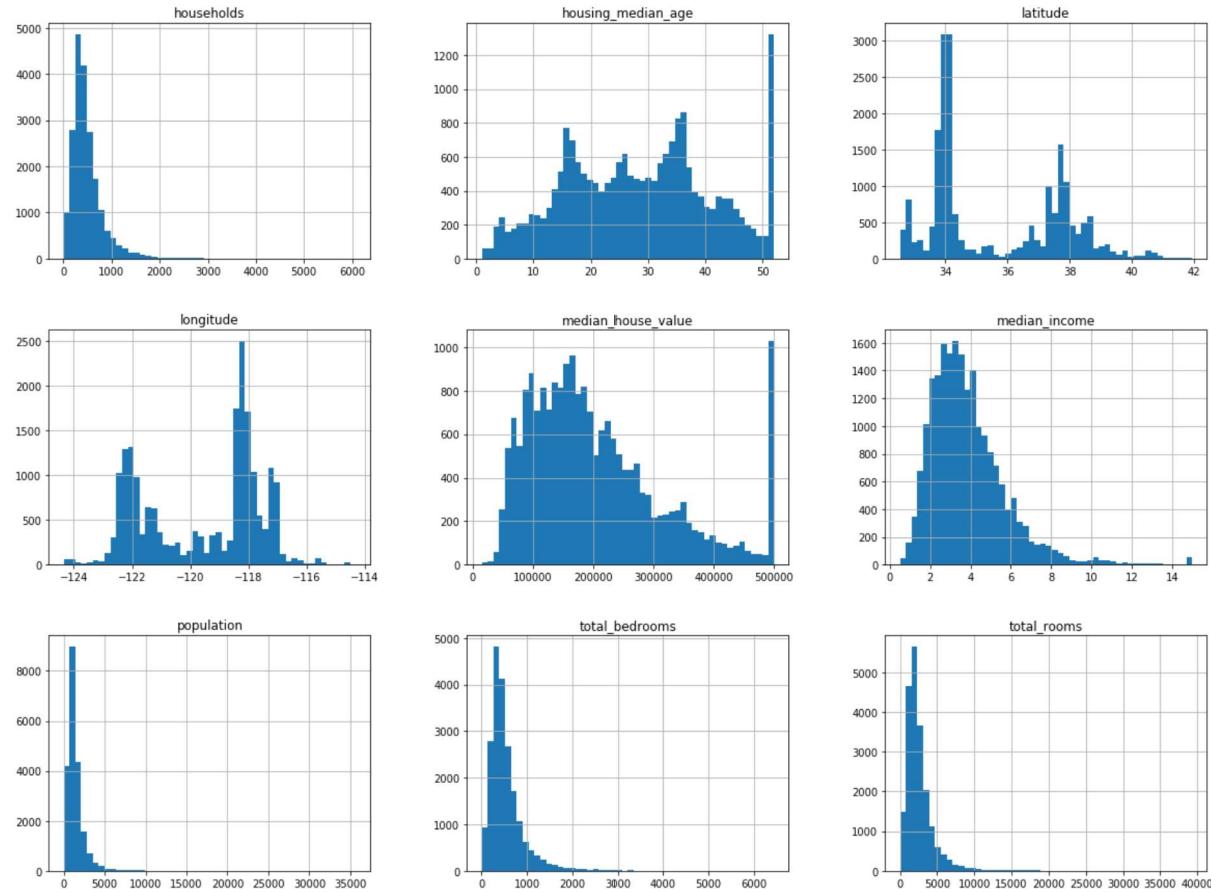
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000

If you want to learn about different ways of accessing elements or other functions it's useful to check out the getting started section [here](https://pandas.pydata.org/pandas-docs/stable/getting_started/index.html) (https://pandas.pydata.org/pandas-docs/stable/getting_started/index.html)

Let's start visualizing the dataset

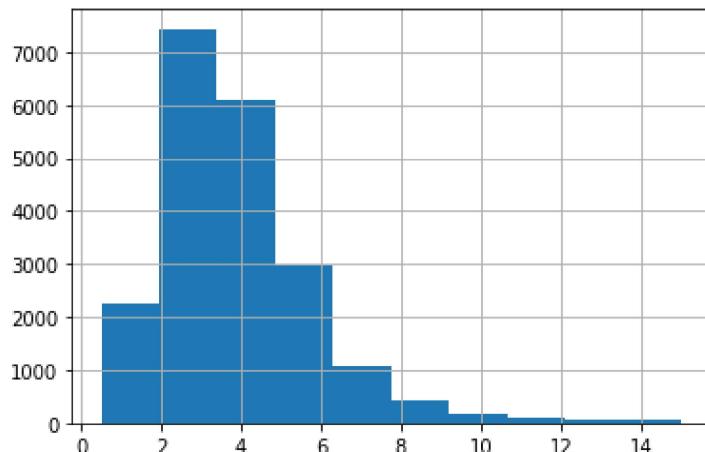
In [10]:

```
# We can draw a histogram for each of the dataframes features  
# using the hist function  
housing.hist(bins=50, figsize=(20,15))  
# save_fig("attribute_histogram_plots")  
plt.show() # pandas internally uses matplotlib, and to display all the figures  
# the show() function must be called
```



In [11]:

```
# if you want to have a histogram on an individual feature:  
housing["median_income"].hist()  
plt.show()
```



We can convert a floating point feature to a categorical feature by binning or by defining a set of intervals.

For example, to bin the households based on median_income we can use the pd.cut function

In [12]:

```
# assign each bin a categorical value [1, 2, 3, 4, 5] in this case.  
housing["income_cat"] = pd.cut(housing["median_income"],  
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],  
                               labels=[1, 2, 3, 4, 5])  
  
housing["income_cat"].value_counts()
```

Out[12]:

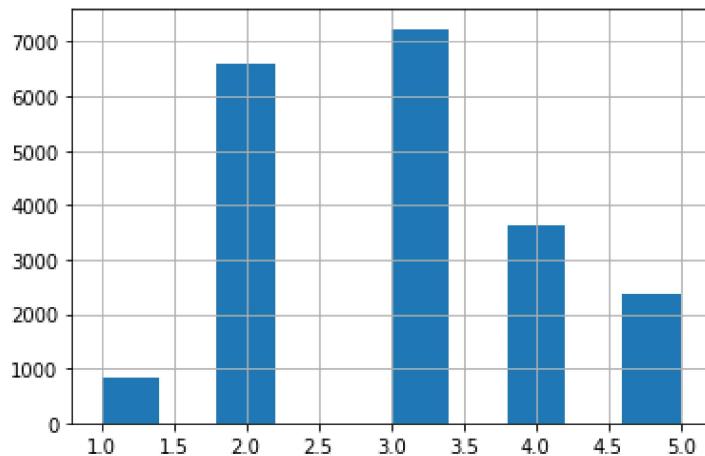
```
3    7236  
2    6581  
4    3639  
5    2362  
1     822  
Name: income_cat, dtype: int64
```

In [13]:

```
housing["income_cat"].hist()
```

Out[13]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x21937d4ab48>
```

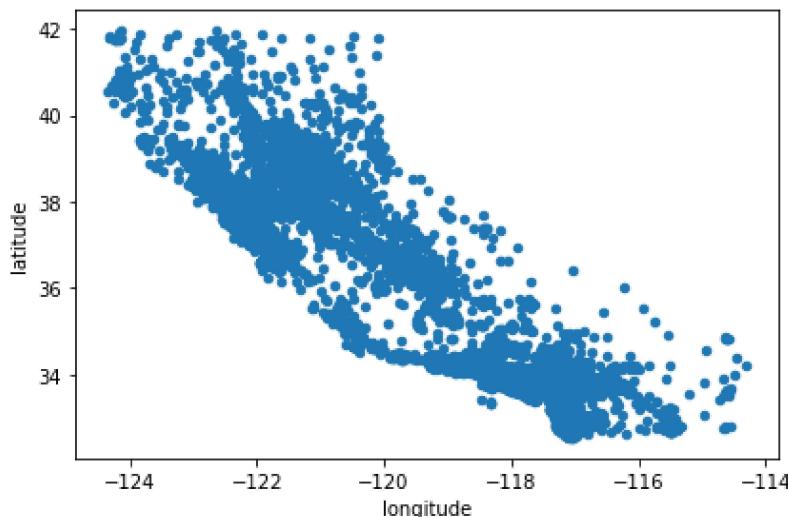


Next let's visualize the household incomes based on latitude & longitude coordinates

In [14]:

```
## here's a not so interesting way plotting it
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

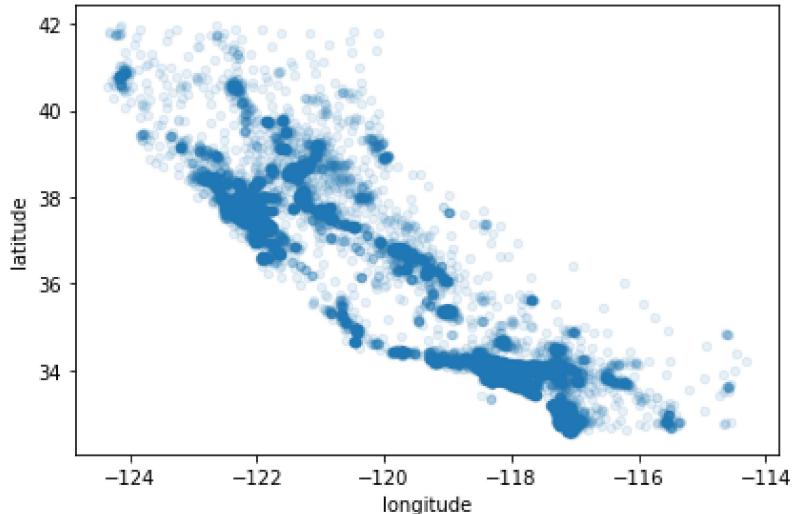
Saving figure bad_visualization_plot



In [15]:

```
# we can make it look a bit nicer by using the alpha parameter,  
# it simply plots less dense areas lighter.  
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)  
save_fig("better_visualization_plot")
```

Saving figure better_visualization_plot



In [16]:

```
# A more interesting plot is to color code (heatmap) the dots
# based on income. The code below achieves this

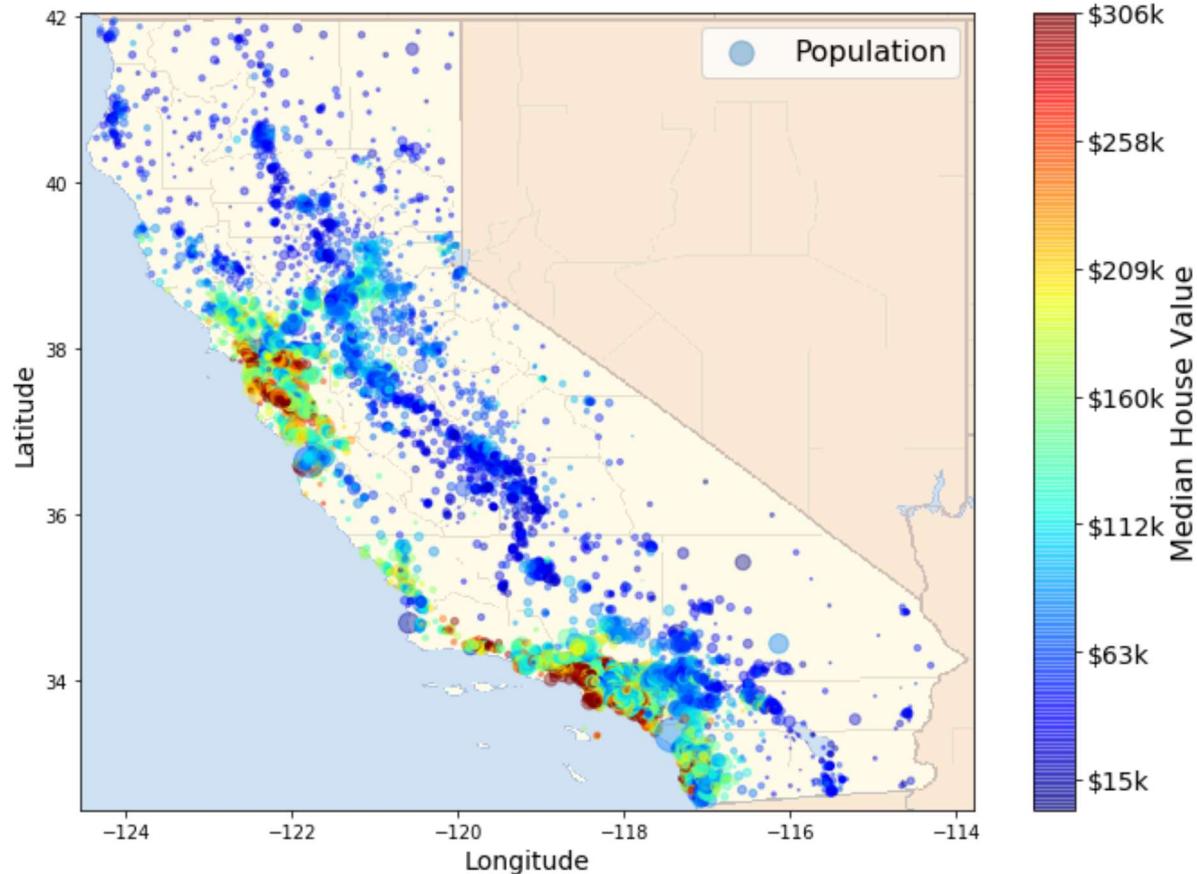
# Load an image of california
images_path = os.path.join('./', "images")
os.makedirs(images_path, exist_ok=True)
filename = "california.png"

import matplotlib.image as mpimg
california_img=mpimg.imread(os.path.join(images_path, filename))
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,7),
                   s=housing['population']/100, label="Population",
                   c="median_house_value", cmap=plt.get_cmap("jet"),
                   colorbar=False, alpha=0.4,
                   )
# overlay the califronia map on the plotted scatter plot
# note: plt.imshow still refers to the most recent figure
# that hasn't been plotted yet.
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.5,
           cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

# setting up heatmap colors based on median_house_value feature
prices = housing["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cb = plt.colorbar()
cb.ax.set_yticklabels(["${:dk}{}".format(v/1000) for v in tick_values], fontsize=14)
cb.set_label('Median House Value', fontsize=16)

plt.legend(fontsize=16)
save_fig("california_housing_prices_plot")
plt.show()
```

Saving figure california_housing_prices_plot



Not surprisingly, the most expensive houses are concentrated around the San Francisco/Los Angeles areas.

Up until now we have only visualized feature histograms and basic statistics.

When developing machine learning models the predictiveness of a feature for a particular target of interest is what's important.

It may be that only a few features are useful for the target at hand, or features may need to be augmented by applying certain transformations.

None the less we can explore this using correlation matrices.

In [17]:

```
corr_matrix = housing.corr()
```

In [18]:

```
# for example if the target is "median_house_value", most correlated features can be sorted
# which happens to be "median_income". This also intuitively makes sense.
corr_matrix[ "median_house_value" ].sort_values(ascending=False)
```

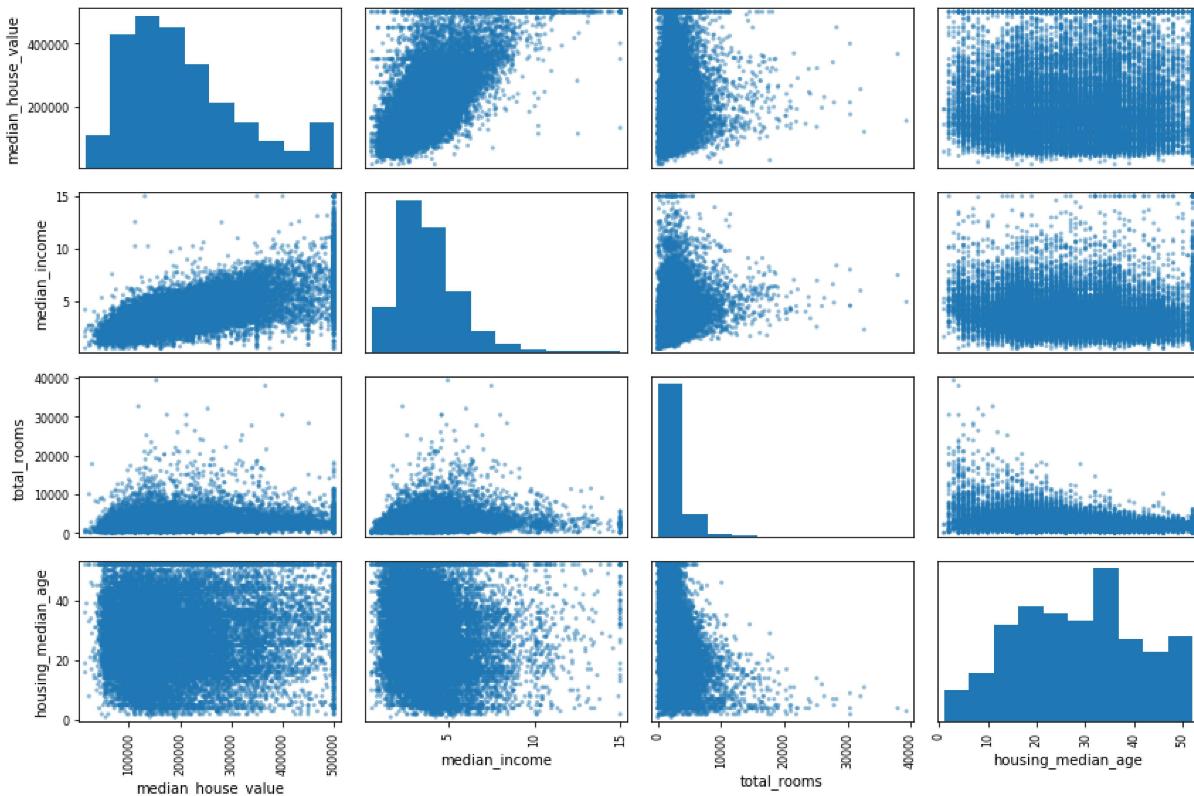
Out[18]:

```
median_house_value      1.000000
median_income          0.688075
total_rooms            0.134153
housing_median_age     0.105623
households             0.065843
total_bedrooms         0.049686
population             -0.024650
longitude              -0.045967
latitude               -0.144160
Name: median_house_value, dtype: float64
```

In [19]:

```
# the correlation matrix for different attributes/features can also be plotted
# some features may show a positive correlation/negative correlation or
# it may turn out to be completely random!
from pandas.plotting import scatter_matrix
attributes = [ "median_house_value", "median_income", "total_rooms",
               "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
save_fig("scatter_matrix_plot")
```

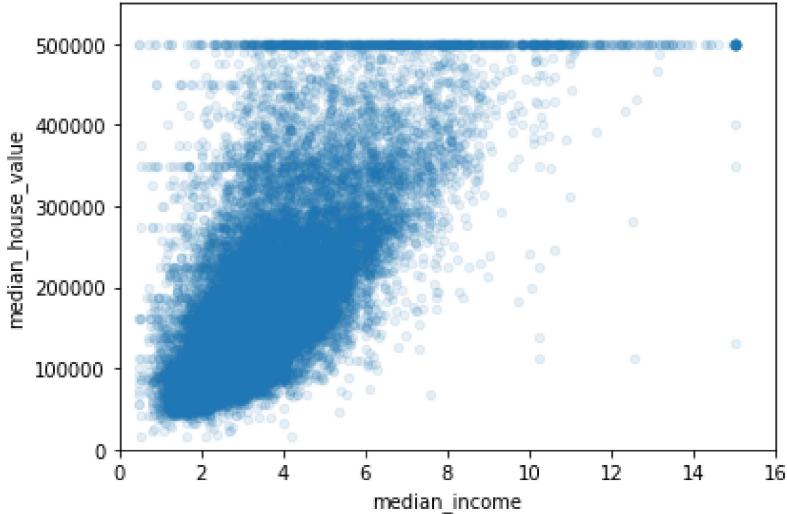
Saving figure scatter_matrix_plot



In [20]:

```
# median income vs median house value plot plot 2 in the first row of top figure
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
plt.axis([0, 16, 0, 550000])
save_fig("income_vs_house_value_scatterplot")
```

Saving figure income_vs_house_value_scatterplot



Augmenting Features

New features can be created by combining different columns from our data set.

- rooms_per_household = total_rooms / households
- bedrooms_per_room = total_bedrooms / total_rooms
- etc.

In [21]:

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"] = housing["population"]/housing["households"]
```

In [22]:

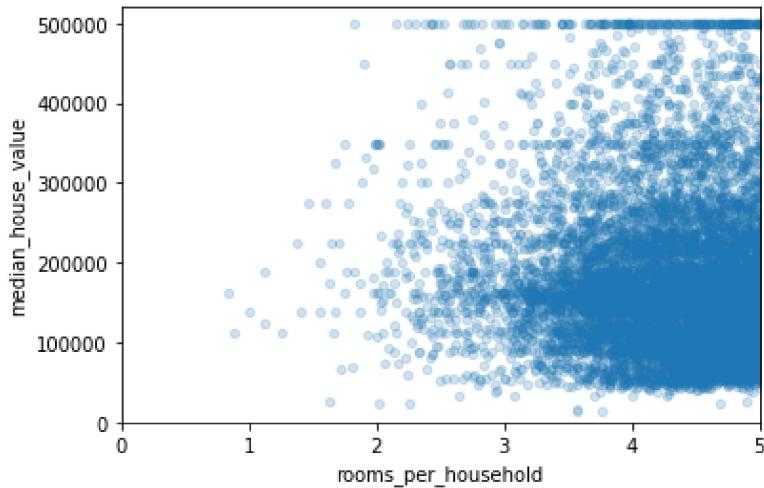
```
# obtain new correlations
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[22]:

```
median_house_value      1.000000
median_income          0.688075
rooms_per_household   0.151948
total_rooms            0.134153
housing_median_age     0.105623
households             0.065843
total_bedrooms         0.049686
population_per_household -0.023737
population              -0.024650
longitude                -0.045967
latitude                  -0.144160
bedrooms_per_room       -0.255880
Name: median_house_value, dtype: float64
```

In [23]:

```
housing.plot(kind="scatter", x="rooms_per_household", y="median_house_value",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```



In [24]:

```
housing.describe()
```

Out[24]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000

Preparing Dataset for ML

Once we've visualized the data, and have a certain understanding of how the data looks like. It's time to clean!

Most of your time will be spent on this step, although the datasets used in this project are relatively nice and clean... it could get real dirty.

After having cleaned your dataset you're aiming for:

- train set
- test set

In some cases you might also have a validation set as well for tuning hyperparameters (don't worry if you're not familiar with this term yet..)

In supervised learning setting your train set and test set should contain (**feature**, **target**) tuples.

- **feature**: is the input to your model
- **target**: is the ground truth label
 - when target is categorical the task is a classification task
 - when target is floating point the task is a regression task

We will make use of [scikit-learn \(<https://scikit-learn.org/stable/>\)](https://scikit-learn.org/stable/) python package for preprocessing.

Scikit learn is pretty well documented and if you get confused at any point simply look up the function/object!

In [25]:

```
from sklearn.model_selection import StratifiedShuffleSplit
# Let's first start by creating our train and test sets
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    train_set = housing.loc[train_index]
    test_set = housing.loc[test_index]
```

In [26]:

```
housing = train_set.drop("median_house_value", axis=1) # drop Labels for training set features
                                                    # the input to the model should not contain the true label
housing_labels = train_set["median_house_value"].copy()
```

Dealing With Incomplete Data

In [27]:

```
# have you noticed when looking at the dataframe summary certain rows
# contained null values? we can't just leave them as nulls and expect our
# model to handle them for us...
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

Out[27]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households
4629	-118.30	34.07		18.0	3759.0	NaN	3296.0
6068	-117.86	34.01		16.0	4632.0	NaN	3038.0
17923	-121.97	37.35		30.0	1955.0	NaN	999.0
13656	-117.30	34.05		6.0	2155.0	NaN	1039.0
19252	-122.79	38.48		7.0	6837.0	NaN	3468.0

In [28]:

```
sample_incomplete_rows.dropna(subset=["total_bedrooms"])      # option 1: simply drop rows that have null values
```

Out[28]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median...
4629	-118.30	34.07	34.0	18.0	3759.0	3296.0	3296.0	1462.0
6068	-117.86	34.01	34.0	16.0	4632.0	3038.0	3038.0	727.0
17923	-121.97	37.35	37.35	30.0	1955.0	999.0	999.0	386.0
13656	-117.30	34.05	34.0	6.0	2155.0	1039.0	1039.0	391.0
19252	-122.79	38.48	38.48	7.0	6837.0	3468.0	3468.0	1405.0

In [29]:

```
sample_incomplete_rows.drop("total_bedrooms", axis=1)          # option 2: drop the complete feature
```

Out[29]:

	longitude	latitude	housing_median_age	total_rooms	population	households	median_income
4629	-118.30	34.07	18.0	3759.0	3296.0	1462.0	2.2708
6068	-117.86	34.01	16.0	4632.0	3038.0	727.0	5.1762
17923	-121.97	37.35	30.0	1955.0	999.0	386.0	4.6328
13656	-117.30	34.05	6.0	2155.0	1039.0	391.0	1.6675
19252	-122.79	38.48	7.0	6837.0	3468.0	1405.0	3.1662

In [30]:

```
median = housing["total_bedrooms"].median()  
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3: replace na values with median values  
sample_incomplete_rows
```

Out[30]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households
4629	-118.30	34.07	18.0	3759.0	433.0	3296.0	1462.0
6068	-117.86	34.01	16.0	4632.0	433.0	3038.0	727.0
17923	-121.97	37.35	30.0	1955.0	433.0	999.0	386.0
13656	-117.30	34.05	6.0	2155.0	433.0	1039.0	391.0
19252	-122.79	38.48	7.0	6837.0	433.0	3468.0	1405.0

Could you think of another plausible imputation for this dataset? (Not graded)

Prepare Data

In [31]:

```
# This cell implements the complete pipeline for preparing the data
# using sklearns TransformerMixins
# Earlier we mentioned different types of features: categorical, and floats.
# In the case of floats we might want to convert them to categories.
# On the other hand categories in which are not already represented as integers must be mapped
# to integers before
# feeding to the model.

# Additionally, categorical values could either be represented as one-hot vectors or simple as
# normalized/unnormalized integers.
# Here we encode them using one hot vectors.

from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

from sklearn.base import BaseEstimator, TransformerMixin

imputer = SimpleImputer(strategy="median") # use median imputation for missing values
housing_num = housing.drop("ocean_proximity", axis=1) # remove the categorical feature
# column index
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

#
class AugmentFeatures(BaseEstimator, TransformerMixin):
    """
    implements the previous features we had defined
    housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
    housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
    housing["population_per_household"] = housing["population"]/housing["households"]
    """

    def __init__(self, add_bedrooms_per_room = True):
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                       bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = AugmentFeatures(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)

num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('attribs_adder', AugmentFeatures()),
    ('std_scaler', StandardScaler()),
])
])
```

```

housing_num_tr = num_pipeline.fit_transform(housing_num)
numerical_features = list(housing_num)
categorical_features = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
    ("num", num_pipeline, numerical_features),
    ("cat", OneHotEncoder(), categorical_features),
])
housing_prepared = full_pipeline.fit_transform(housing)

```

Select a model and train

Once we have prepared the dataset it's time to choose a model.

As our task is to predict the median_house_value (a floating value), regression is well suited for this.

In [32]:

```

from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)

# Let's try the full preprocessing pipeline on a few training instances
data = test_set.iloc[:5]
labels = housing_labels.iloc[:5]
data_prepared = full_pipeline.transform(data)

print("Predictions:", lin_reg.predict(data_prepared))
print("Actual labels:", list(labels))

```

Predictions: [425717.48517515 267643.98033218 227366.19892733 199614.48287493
161425.25185885]

Actual labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]

We can evaluate our model using certain metrics, a fitting metric for regression is the mean-squared-loss

$$\$L(\hat{y}, Y) = \sum_i^N (\hat{y}_i - y_i)^2 \$$$

where \hat{y} is the predicted value, and y is the ground truth label.

In [33]:

```

from sklearn.metrics import mean_squared_error

preds = lin_reg.predict(housing_prepared)
mse = mean_squared_error(housing_labels, preds)
rmse = np.sqrt(mse)
rmse

```

Out[33]:

67784.32202861732

TODO: Applying the end-end ML steps to a different dataset.

We will apply what we've learnt to another dataset (airbnb dataset). We will predict airbnb price based on other features.

[25 pts] Visualizing Data

[5 pts] Load the data + statistics

- load the dataset
- display the first few rows of the data
- drop the following columns: name, host_id, host_name, last_review
- display a summary of the statistics of the loaded data
- plot histograms for 3 features of your choice

In [34]:

```
def load_airbnb_data(airbnb_path):
    csv_path = os.path.join(airbnb_path, "AB_NYC_2019.csv")
    return pd.read_csv(csv_path)
```

In [35]:

```
AIRBNB_PATH = os.path.join("datasets", "airbnb")
airbnb = load_airbnb_data(AIRBNB_PATH)

airbnb.head()
```

Out[35]:

	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude	longit
0	2539	Clean & quiet apt home by the park	2787	John	Brooklyn	Kensington	40.64749	-73.97
1	2595	Skylit Midtown Castle	2845	Jennifer	Manhattan	Midtown	40.75362	-73.98
2	3647	THE VILLAGE OF HARLEM....NEW YORK !	4632	Elisabeth	Manhattan	Harlem	40.80902	-73.94
3	3831	Cozy Entire Floor of Brownstone	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514	-73.95
4	5022	Entire Apt: Spacious Studio/Loft by central park	7192	Laura	Manhattan	East Harlem	40.79851	-73.94

In [36]:

```
airbnb = airbnb.drop(columns=["name", "host_id", "host_name", "last_review"])
airbnb.describe()
```

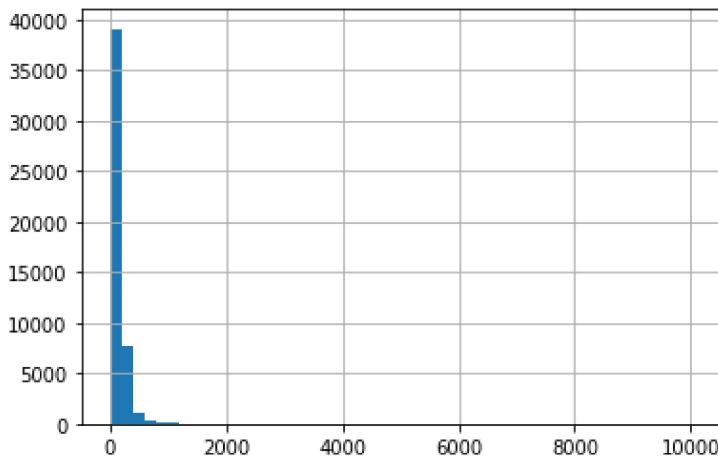
Out[36]:

	id	latitude	longitude	price	minimum_nights	number_of_reviews	i
count	4.889500e+04	48895.000000	48895.000000	48895.000000	48895.000000	48895.000000	48895.000000
mean	1.901714e+07	40.728949	-73.952170	152.720687	7.029962	23.274466	23.274466
std	1.098311e+07	0.054530	0.046157	240.154170	20.510550	44.550582	44.550582
min	2.539000e+03	40.499790	-74.244420	0.000000	1.000000	0.000000	0.000000
25%	9.471945e+06	40.690100	-73.983070	69.000000	1.000000	1.000000	1.000000
50%	1.967728e+07	40.723070	-73.955680	106.000000	3.000000	5.000000	5.000000
75%	2.915218e+07	40.763115	-73.936275	175.000000	5.000000	24.000000	24.000000
max	3.648724e+07	40.913060	-73.712990	10000.000000	1250.000000	629.000000	629.000000

◀ ▶

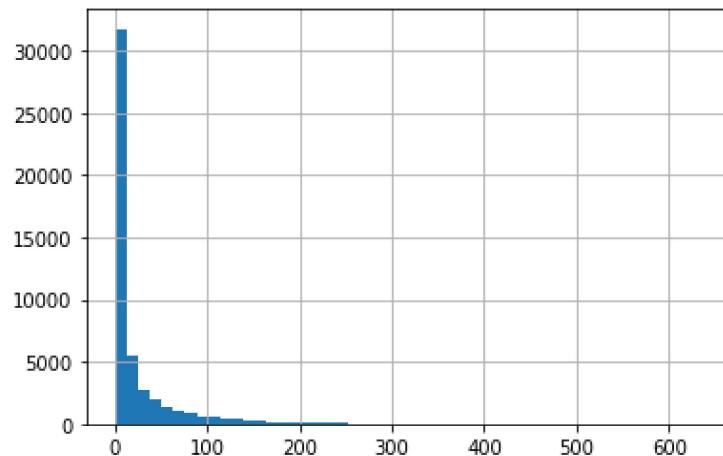
In [37]:

```
airbnb["price"].hist(bins=50)
plt.show()
```



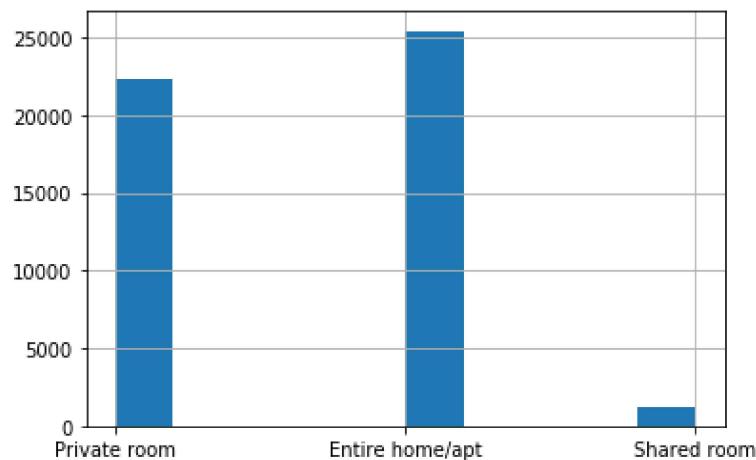
In [38]:

```
airbnb["number_of_reviews"].hist(bins=50)  
plt.show()
```



In [39]:

```
airbnb["room_type"].hist()  
plt.show()
```



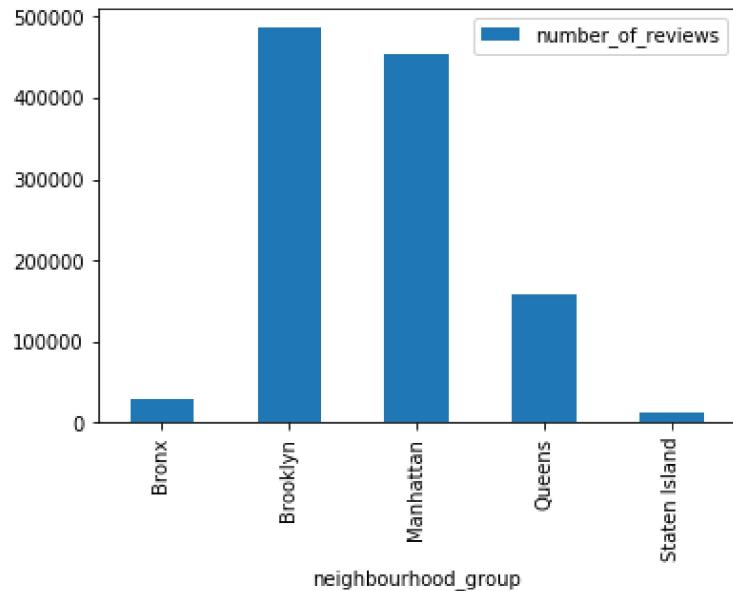
[5 pts] Plot total number_of_reviews per neighbourhood_group

In [40]:

```
airbnb[["neighbourhood_group", "number_of_reviews"]].groupby("neighbourhood_group").sum().plot.  
bar()
```

Out[40]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2193c4b4c08>
```

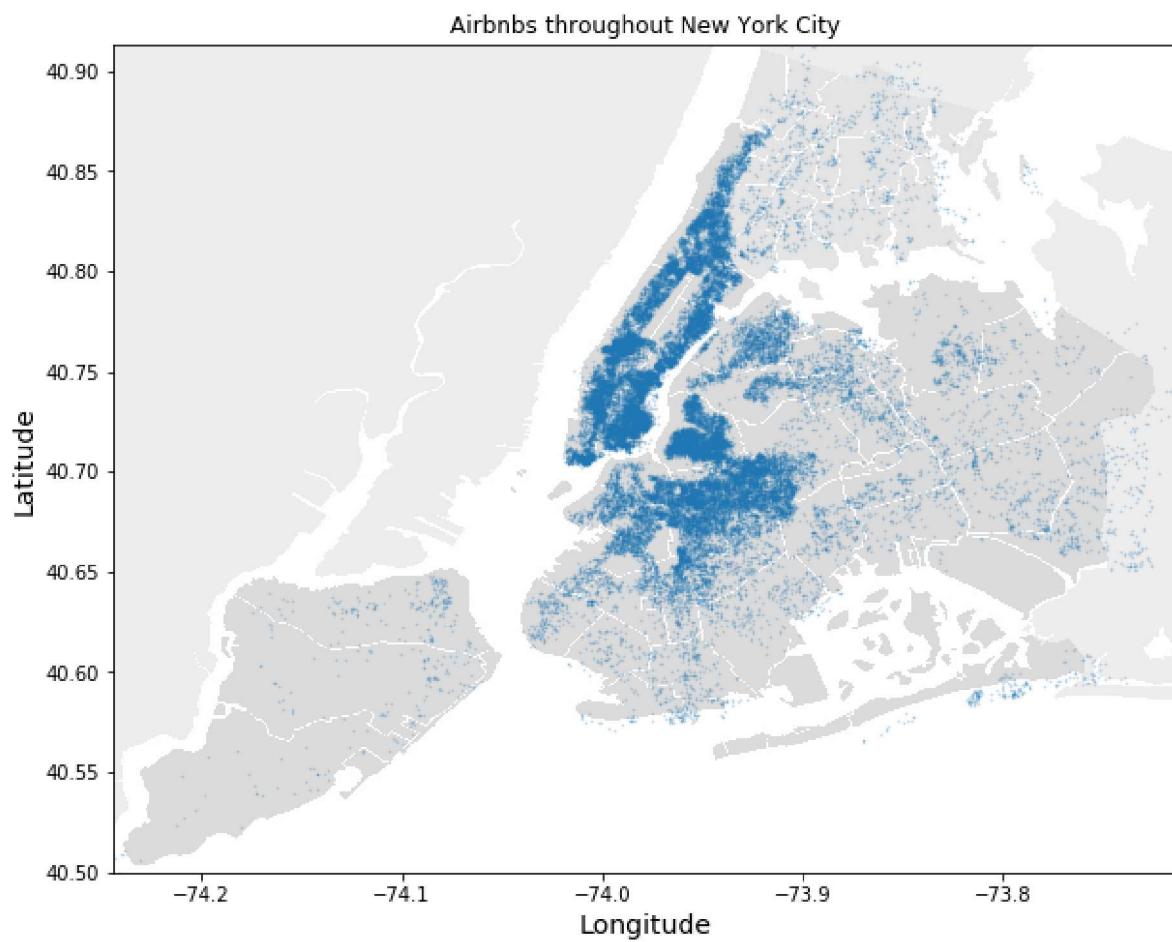


[5 pts] Plot map of airbnbs throughout New York (if it gets too crowded take a subset of the data, and try to make it look nice if you can :)).

In [41]:

```
ny_map_path = os.path.join(images_path, "ny.png")
ny_img = mpimg.imread(ny_map_path, 0)

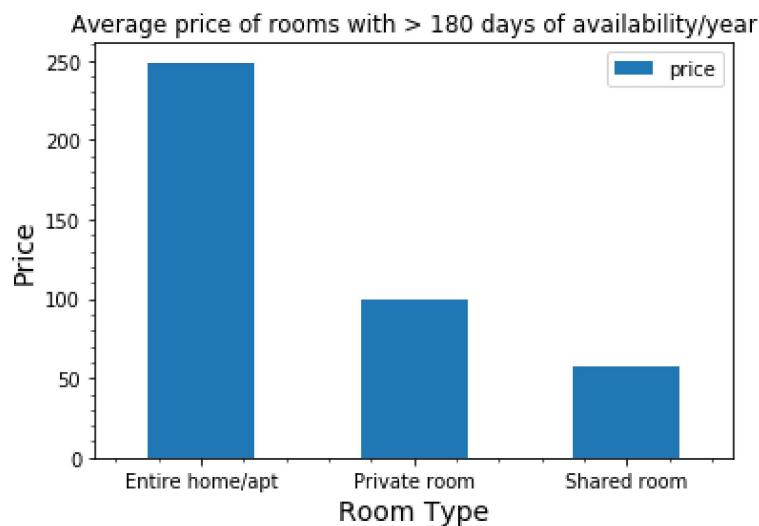
ax2 = airbnb.plot(kind="scatter", x="longitude", y="latitude", alpha=0.25, figsize=(15,8), s=0.25)
plt.imshow(ny_img, extent=[-74.244420, -73.712990, 40.499790, 40.913060], alpha=0.25)
plt.xlabel("Longitude", fontsize=14)
plt.ylabel("Latitude", fontsize=14)
plt.title("Airbnbs throughout New York City")
plt.show()
```



[5 pts] Plot average price of room types who have availability greater than 180 days.

In [42]:

```
available_rooms = airbnb.loc[airbnb["availability_365"] > 180]
ax3 = available_rooms[["room_type", "price"]].groupby("room_type").mean().plot.bar(rot=0)
ax3.set_ylabel("Price", fontsize=14)
ax3.set_xlabel("Room Type", fontsize=14)
ax3.set_title("Average price of rooms with > 180 days of availability/year")
ax3.minorticks_on()
```



[5 pts] Plot correlation matrix

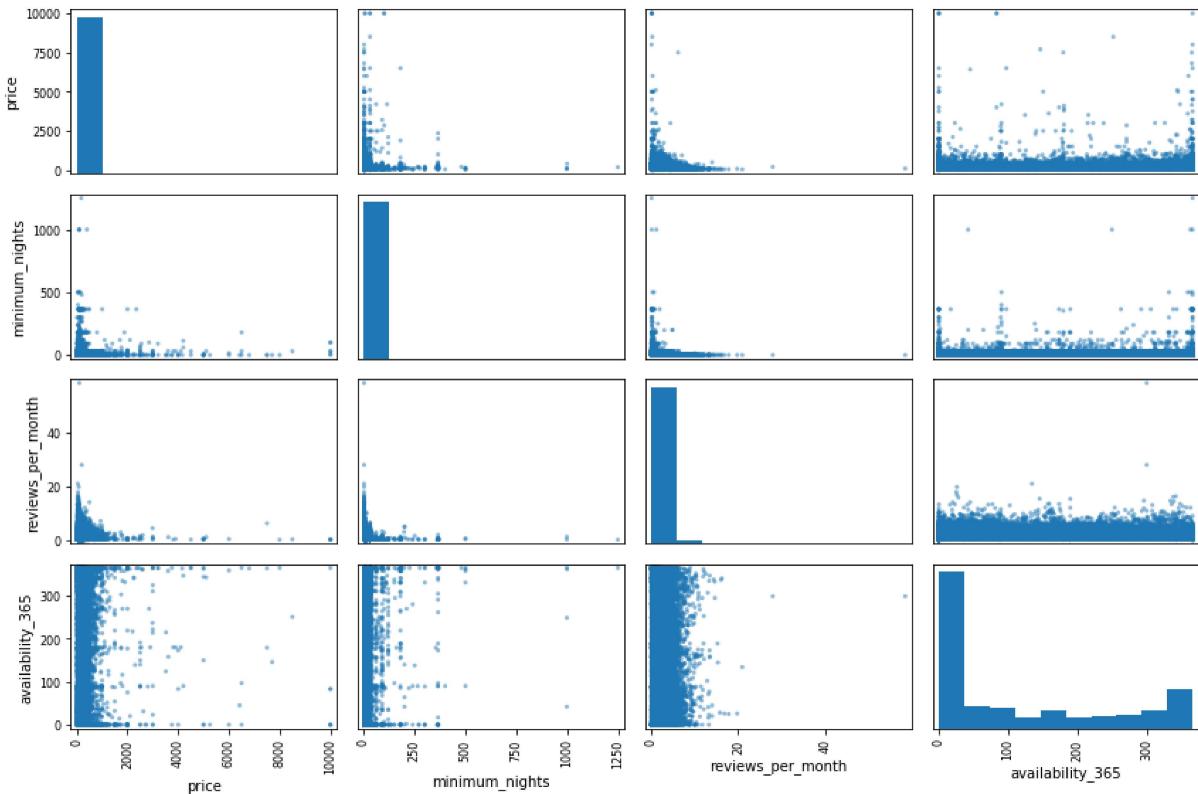
- which features have positive correlation?
- which features have negative correlation?

In [43]:

```
airbnb_corr_mat = airbnb.corr()
airbnb_attributes = ["price", "minimum_nights", "reviews_per_month", "availability_365"]
scatter_matrix(airbnb[airbnb_attributes], figsize=(12,8))
save_fig("airbnb_scatter_matrix_plot")

# min nights vs. price; reviews per month vs. price; and min nights vs. reviews per month seem
# to have a negative correlation.
# all others seem to have no correlation.
```

Saving figure airbnb_scatter_matrix_plot



[25 pts] Prepare the Data

Note: Order of augment/impute/splitting swapped to simplify code

[5 pts] Augment the dataframe with two other features which you think would be useful

In [44]:

```
airbnb["host_listing_availability"] = airbnb["calculated_host_listings_count"] * airbnb["availability_365"]
airbnb["max_bookings_365"] = airbnb["availability_365"] / airbnb["minimum_nights"]
airbnb.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48895 entries, 0 to 48894
Data columns (total 14 columns):
id                      48895 non-null int64
neighbourhood_group      48895 non-null object
neighbourhood            48895 non-null object
latitude                 48895 non-null float64
longitude                48895 non-null float64
room_type                48895 non-null object
price                    48895 non-null int64
minimum_nights            48895 non-null int64
number_of_reviews          48895 non-null int64
reviews_per_month          38843 non-null float64
calculated_host_listings_count 48895 non-null int64
availability_365           48895 non-null int64
host_listing_availability 48895 non-null int64
max_bookings_365           48895 non-null float64
dtypes: float64(4), int64(7), object(3)
memory usage: 5.2+ MB
```

[5 pts] Impute any missing feature with a method of your choice, and briefly discuss why you chose this imputation method

In [45]:

```
# NaN reviews per month seem to be created when number_of_reviews == 0.
# Thus I chose to impute reviews_per_month by replacing NaN's with 0's.
airbnb["reviews_per_month"].fillna(value=0.0, inplace=True)
airbnb["reviews_per_month"].fillna(value=0.0, inplace=True)

airbnb.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48895 entries, 0 to 48894
Data columns (total 14 columns):
id                      48895 non-null int64
neighbourhood_group      48895 non-null object
neighbourhood            48895 non-null object
latitude                 48895 non-null float64
longitude                48895 non-null float64
room_type                48895 non-null object
price                    48895 non-null int64
minimum_nights            48895 non-null int64
number_of_reviews          48895 non-null int64
reviews_per_month          48895 non-null float64
calculated_host_listings_count 48895 non-null int64
availability_365           48895 non-null int64
host_listing_availability 48895 non-null int64
max_bookings_365           48895 non-null float64
dtypes: float64(4), int64(7), object(3)
memory usage: 5.2+ MB
```

[5 pts] Set aside 20% of the data as test test (80% train, 20% test).

In [46]:

```
from sklearn.model_selection import train_test_split
airbnb_train, airbnb_test = train_test_split(airbnb, test_size = 0.2, random_state=24)
airbnb_X_train = airbnb_train.drop("price", axis=1).copy()
airbnb_y_train = airbnb_train["price"].copy()
airbnb_X_test = airbnb_test.drop("price", axis=1).copy()
airbnb_y_test = airbnb_test["price"].copy()
```

[10 pts] Code complete data pipeline using sklearn mixins

In [47]:

```
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

from sklearn.base import BaseEstimator, TransformerMixin

# remove identifying/categorical data. "name", "host_id", "host_name", "last_review" dropped previously.
airbnb_num = airbnb_X_train.drop(["neighbourhood_group", "neighbourhood", "room_type"], axis=1)

airbnb_num_features = list(airbnb_num)
# "name", "host_id", "host_name", "last_review" dropped previously.
airbnb_cat_features = ["neighbourhood_group", "neighbourhood", "room_type"]

airbnb_full_pipeline = ColumnTransformer([
    ("num", StandardScaler(), airbnb_num_features),
    ("cat", OneHotEncoder(handle_unknown='ignore'), airbnb_cat_features),
])
airbnb_train_prepared = airbnb_full_pipeline.fit_transform(airbnb_X_train)
```

[15 pts] Fit a model of your choice

The task is to predict the price, you could refer to the housing example on how to train and evaluate your model using MSE. Provide both test and train set MSE values.

In [48]:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(airbnb_train_prepared, airbnb_y_train)
# Run full preprocessing pipeline on training instances
from sklearn.metrics import mean_squared_error

train_preds = lin_reg.predict(airbnb_train_prepared)
train_mse = mean_squared_error(airbnb_y_train, train_preds)
print("Train mse:", train_mse)

airbnb_test_prepared = airbnb_full_pipeline.transform(airbnb_X_test)
test_preds = lin_reg.predict(airbnb_test_prepared)
test_mse = mean_squared_error(airbnb_y_test, test_preds)
print("Test mse:", test_mse)
```

Train mse: 51745.6308252567

Test mse: 47123.45715177272

In []: